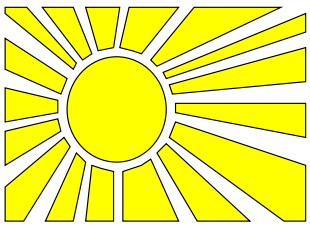
EPCC-SS-2001-06

A Java Grande benchmark to compare Java and C++ performance for a discrete event simulation application

Ana Bosque Arbiol

Abstract

The aim of this project is just to study C++ and Java performance for a discrete event simulator. We are interested in introducing this kind of applications in the suite of benchmark tests which are in the Java Grande Benchmark. In the development of this project we have used the ns simulator, which simulates computer networks. We worked only with the core of this simulator, which takes care of the treatment of events. In this part of code, a structure called the scheduler stores future events and decides which is the next to be executed. We analysed the implementation of this structure and its performance inside the simulator in order to separate it from the whole ns code. To do this we executed the ns simulator with different computer networks models. To execute the scheduler structure out of the simulator an event generator which simulates the performance of the one that has the simulator has to be built. We did this using trace files due to the complex distribution which were obtained in the execution of the simulator for the event generator. These trace files were obtained in the execution of the ns simulator. The first model was written in C++ because the ns simulator is written in this language. When we verified that the events generated were exactly the same as in the ns simulator, we translated this model into Java and executed the two models in different environments and platforms. As a result of these executions we obtained a better performance in Java than in C++, specially when the size of the problem is increased.



1 Introduction

Java is an object-oriented programming language and Grande applications those which have large memory, network or computational requirements. Java wasn't designed for these type of applications. Nowadays the use of Java has departed from its original design goals. This trend is due to some interesting features that Java has: portability, no need of complex makefiles, support of remote method invocation, remote file access and database access. But Java has also some disadvantages compared with other languages like C or Fortran. These disadvantages include the lack of some features for scientific calculations (nonexistence of complex numbers as a type and multidimensional arrays), the absence of familiar parallel programming models (MPI and OpenMP) and performance.

This last of Java disadvantages is perhaps the most important. Some programmers avoid the use of Java because of this, without considering the last advances in just-in-time (JIT) and adaptive compilers. This behavior is due to the lack of studies and publications relevant to Grande applications in Java. The benchmarking effort made with the Java Grande Benchmark [**?**, **?**] tries to address this problem by providing the community a standard benchmark suite which can be used to demostrate the use of Java for Grande applications, to provide metrics for comparing environments in order to inform the users about the most suitable one for their needs and to expose some interesting features in the performance of Grande applications that can guide the development of the environments in appropriate directions.

A discrete event simulation application is a simulator which works with a set of initial events, generating new events or cancelling the old ones during the execution of each event. In this project we worked with the ns simulator, a computer networks simulator. In a computer network when an event is executed (e.g. send a packet) new events can be generated (e.g. switch on a timer, reception of the packet) and others cancelled (e.g. timers which have been switched on before). For further information about this simulator see Appendix 5.1 in which we describe it and give further references. Internet sites about it.

In this project we have built a benchmark to study the performance of an application which inserts and deletes events from a scheduler. This benchmark would fit in Section III of the Java Grande Benchmark [**?**] because it is an application.

We have worked with ns as a base, but for our purpose (comparing C++ and Java performance) it is not neccesary to work with the whole code of ns, because it is a huge code and would take a long time to translate in Java (the code of ns is actually in C and in C++), and to execute the simulations that we are interested in. Thus we have extracted the simulator core which performs the treatment of the events, and ignored the rest. We were interested only in this core because it is the only part of the execution which does not change from one simulation to another. ns allows the building of a huge amount of different computer networks and it is sure than the events generated and cancelled in any of them would be really different to the events in the others. But the scheduler will be always exactly the same: it will insert, cancel and look for the events in the same way, meanwhile the rest is model dependent. Moreover it's possible to change the simulator into another which simulates something different from computer networks and as it was a discrete event application it would be possible to use the same scheduler.

To run this core, we have to create an event generator, an application which inserts the events in the structure and decides what actions to take on the execution of each event. This event generator has to reproduce the same conditions that the simulator creates in its execution. Only

when we have completed this will be translated our model into Java and begin the comparisons between C++ and Java performance.

The remainder of this paper is structured as follows: Section 2 describes the different steps that we took in order to arrive to the final results. Section 3 shows the results that we obtained when we executed the model in C++ and in Java. Section 4 provids some conclusions and Section 5 contains some appendices with further information about some topics related to this project.

2 Our work

In this section we will describe the steps taken, explain the problems that we found and how we solved them. In this report we describe these steps in the same order as they were taken, to help the reader understand what our problems were and how we solved them.

2.1 How the schedule structure works

In the introduction we explained that we did not require the whole code of ns to compare the two languages we are interested in; we only need a part of the ns code called the scheduler. This part treats the events in the simulator: it is a virtual clock which is used to determine when the events have to been executed and a structure in which stores all the events that it has to execute in the future.

The first thing that we did was to understand how the scheduler works in order to isolate it from the rest of the ns simulator. There were different implementations of the scheduler in the ns code: a ListScheduler which consists of a list of events linked by pointers, a HeapScheduler which keeps all the events in a heap and the CalendarScheduler which was a kind of hash list. At first we were only interested in the third structure, the CalendarScheduler, as it is the one that ns uses by default and it has the best theoretical execution time.

The CalendarScheduler defines the operations of insertion, cancellation, looking for an specific event and looking for the next event to execute. It uses a structure which consists of an array of lists, called buckets. Each of these lists are ordered from the first event to execute in the list to the last.

An event has to been inserted in a specific bucket depending on the value of the timestamp. The first question to solve is how to determine the bucket in which an event is inserted. The optimal situation is to have the minimum number of events in each of the lists, because the cost of the operations made over a list are linear in the number of items of the list. To achieve this aim, two actions are taken. The first is to try to balance the load of the structure in order to have the same number of events in every list. The second is to resize the array depending on the number of events that the whole structure has.

To get a balanced load of the schedule structure, it was noted that the most of the events may be generated around some determined points in the time, so if the intervals of the buckets were large, some buckets will contain a lot of events and others will be a very few or even zero. But if the intervals are too small, then a large number of buckets will be required. For this reason circular array is used, granting an unlimited number of buckets. The width of each bucket is calculated depending on the distance between the events in the structure, to avoid the existence of a bucket in which lots of events will be inserted.

The other idea to maintain a small number of events in each of the buckets is to resize the structure depending on the number of events in it. A good load in this structure is to have the number of events between the half and double the number of buckets. If the number of events is more than double the number of buckets, the number of buckets will be doubled and if the number of events is less than the half the number of buckets, the number of buckets will be halfed. These operations of resizing are expensive because they have to relocate all the events in the structure.

The next questions are to know how the different operations are carried out. To insert an event, we calculate the number of bucket in which it has to been inserted, and it is inserted in the list of this bucket in order. To cancel an event we calculate the bucket in which it has to be and it is deleted from the list. To look for the next event to execute it is necessary more information because at first sight all the buckets have the same probability of containing the next event to execute, so we would have to go through the whole array comparing the timestamps of the first event of each bucket. There is a variable in the scheduler which keeps the number of the bucket that contained the last event executed, is that the last event fired, and an interval in time in which was this last event. This interval is built beginning from the timestamp of the last event adding a 1.5*width for the upper limit (this ensures that the upper limit is a point at least in the next bucket) and subtracting a multiple of width for the lower limit (it ensures that the lower limit is a point at least in the previous bucket). When the next event is looked for, the first bucket to check is the one pointed by the variable of the scheculer. If there is an event in this bucket is checked if it is inside the interval or not. If it is inside the interval, it will be the next event fired. If it is outside of the interval or the bucket is empty, the variable and interval will be changed in order that they point to the next bucket in the scheduler and the first comparison will be repeated. This operation will be repeated until the next event will be found. To optimize this operation in case that the separation between events were huger than a complete round in the structure (number of buckets multiplied by the width), after the first round without a positive result it is done a round in the structure comparing the smallest event in each bucket and deciding in this way the next event to be executed.

To obtain more information about the scheduler we decided to run some simulations on ns and to study them. At first we took a very simple one, shown in Appendix 5.2. This simulation was not very helpful because its behaviour is a little strange: at the middle of the execution the load of the scheduler changes drastically, but there wasn't any reason. This network has only two nodes connected by a full duplex connection which send messages one to the other by a tcp protocol. This network was very difficult to scale in order to study the behaviour of the scheduler with different loads because its structure is so simple.

In Appendix 5.3 we show which simulation we used for the benchmark. The network which is simulated in this script is a ring of nodes in which we can vary the number of nodes, the number of web clients, the number of tcp clients, the distance in nodes between servers, et. It is a simulation that we could easily scale, so it was very suitable for studying the behaviour of the scheduler. We simulated it with different numbers of nodes in the ring and studied the time spent in the operations of the scheduler versus the time spent in event executions. We obtained the results shown in Table **??**.

These rates were measured in a 18-processor Sun HPC 400MHz UltraSPARC-II in which there were other users, so they are not very accurate, but we can see clearly that the time spent in the operations of the scheduler increases with the size of the scheduler (If the number of nodes is larger, then there will be more traffic. This means that the number of events will be larger, so

Number of nodes	Scheduler time
16	21.8%
32	22.5%
128	28.0%
256	34.2%
1024	32.7%
2048	64.2%

Table 1: Percentage of execution time spent in the scheduler

the data structure to keep them must be also larger.)

2.2 Which distribution we find in the simulations

When we analized and understood how the scheduler works and we separated it from the rest of the ns simulator, we needed to create an event generator. This generator had to simulate the real simulator, in order for the results to be representative. We therefore analysed the distribution of event generation in the real scheduler.

We ran the simulation described in the previous section and analysed the event distribution with three different sizes of the network, in order to avoid working with special cases. We chose networks of 16, 256 and 2048 nodes. In the previous section we saw that the scheduler time was different from one of these simulations to an other.

We were interested in different aspects of the distribution. One of them was the load of the scheduler in order to know if the structure was resized in the execution of these simulations. If this happens it will mean that the number of events changes a lot in the execution due the existence of periods of time in which there are a lot of cancellations and others with a lot of insertions. We output the number of events in the structure in each insertion and plotted this data. We saw that the shape of the graph was exactly the same in all the simulations: The number of events is the only thing that changes. We also proved that the scheduler is only resized near the start, but never after that. In Figure **??** we can see that at the beginning of the simulation the number of events is increasing all the time, but after that it reaches a point in which the number of event is always around a constant value, and it never reaches a number that would cause the structure be resized. In the simulation of Figure **??** the number of buckets is 128, and since the number of events never reaches 64 or 256, the structure is never resized.

The other aspect we were interested in was the insertion delay distribution. By insertion delay we mean the time between the point in which the event is inserted and the point in which it must be executed. We needed this information because we must reproduce it with our model. Figure **??** shows the graph for the simulation of 16 nodes (the X-axis shows the delay and the Y-axis the frequency; the number of events with a specific delay). Here we only will show graphs of this simulation (16 nodes) because the other simulations had exactly the same shape, differring only in the number of events.

In Figure **??** we can see that there are some peak points and perhaps a wider distribution in the interval [0, 1] because we can see that there are a lot of points there. To see this distribution more clearly we only plotted the interval [0, 1] and removed the peak points. Figure **??** shows the graph with this changes. In this graph we can see three different distributions: one around zero,

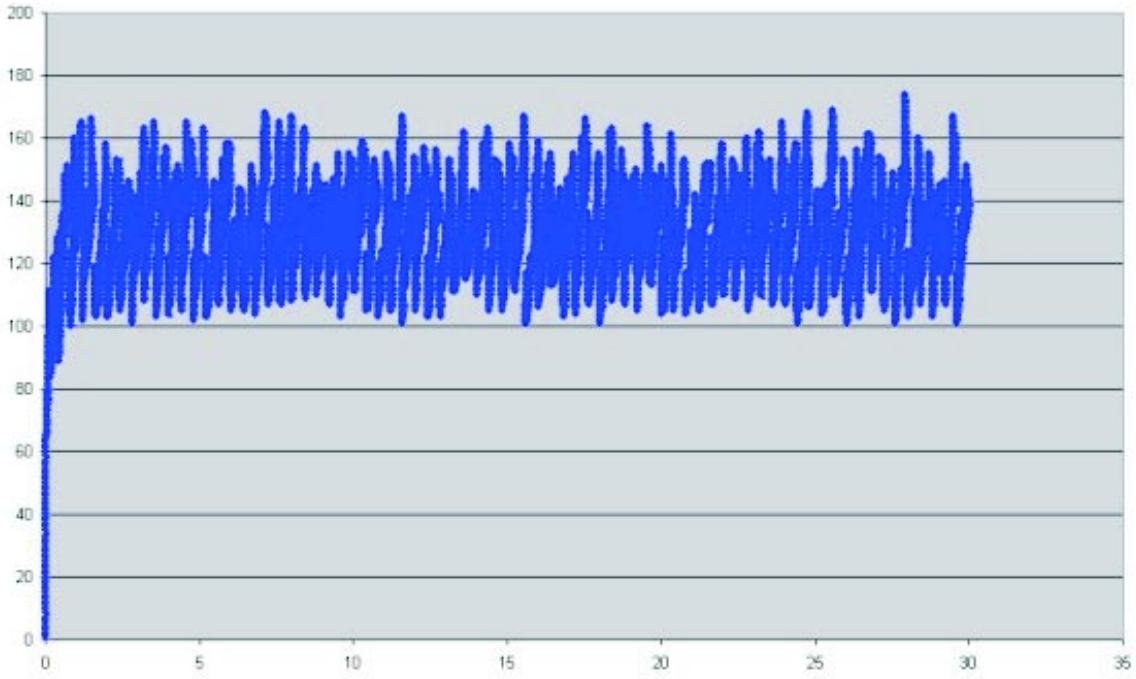


Figure 1: Load of the scheduler in the simulation of a network with 16 nodes

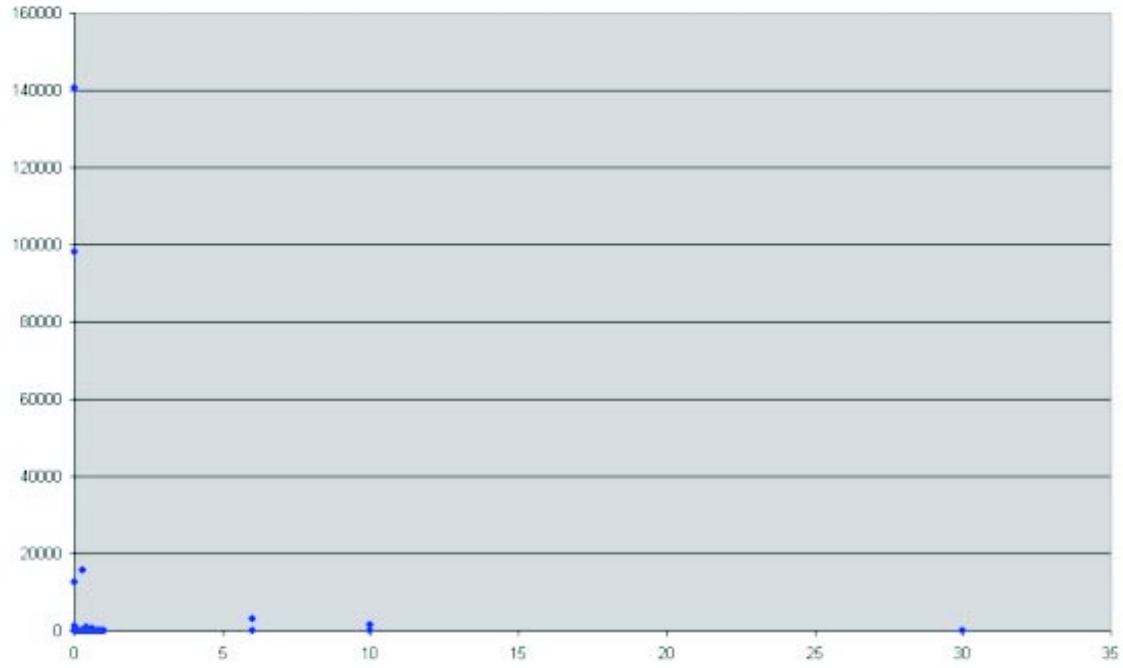


Figure 2: Delays insertion distribution for the simulation of 16 nodes

another in the interval [0, 0.4] and another in [0.4, 1]. Figure **??** shows us these distributions more clearly. Within these distributions we do not observe find any more structive.

We were also interested in the number of peak points that appeared in this distribution, so we

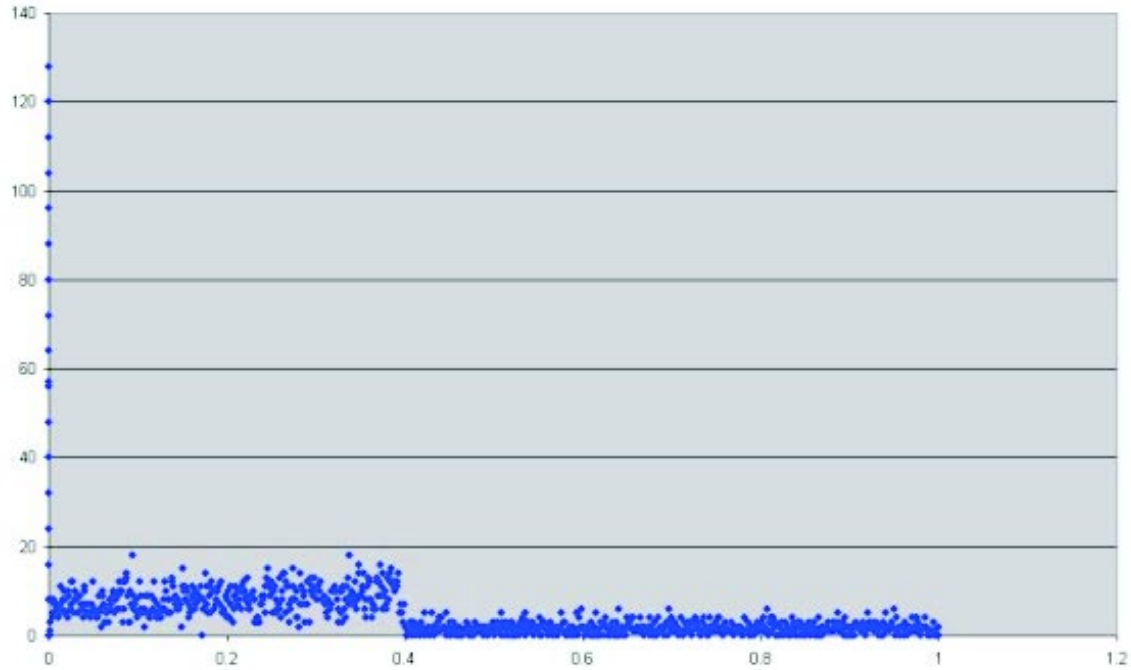


Figure 3: Delays insertion distribution between 0 and 1 without peak points

produced the graph in Figure **??** to count them. We can see that the peak points around zero are significantly larger than the rest of them.

The last aspect that we analysed is the cancel delay distribution. By cancel delay we mean the time between time in which the event is cancelled and the point in which it had to be executed. Figure **??** shows this; the X-axis shows the delay and the Y-axis the frequency, (the number of events with a specific delay). In this graph we can see that there are events cancelled with almost every delay, and also a lot of peaks points

2.3 How our model works

At first we wanted to build an event generator that simulates the one in the ns simulator. This was the reason for the study described in the previous section. But after analysing the data we realized that there was no familiar distribution. Moreover, there was more than one distribution. For these reasons we decided that it was not feasible to build the event generator without any additional information because it would be very difficult to demostrate that it behaved like a real simulation. We decided to create trace files from the ns simulator and our model will read them to decide what it will do in the execution of each event.

In the trace files we have to record which operations have to be executed in the execution of an event, that is, how many insertions (with their delays), how many cancellations (with enough information to locate these events) and how many searches (with the identification of the events searched) are done by the network component which handles the event. All this information is easy to obtain from the execution of the different simulations. In Appendix 5.4 we explain in more detail the data recorded in the trace files.

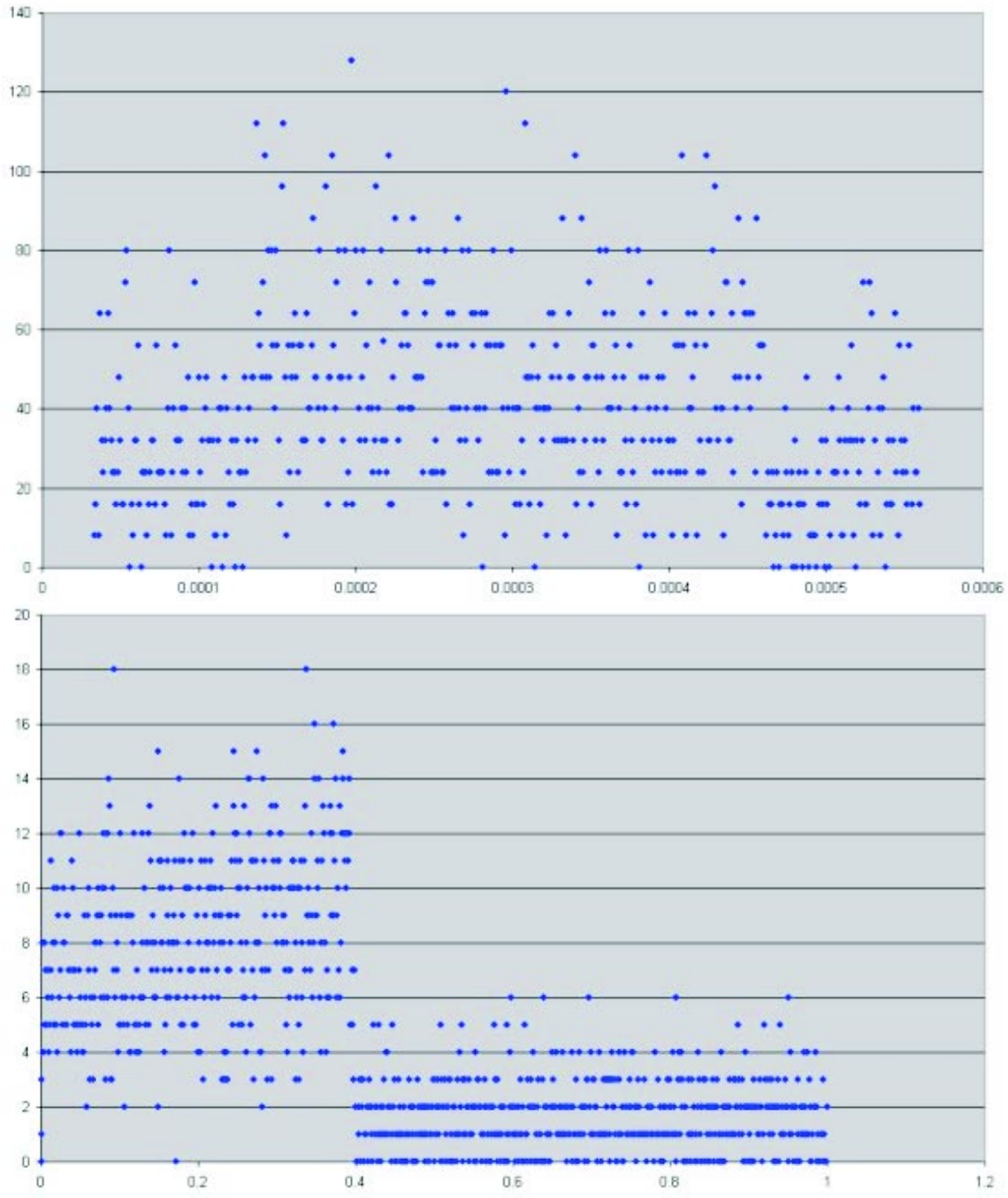


Figure 4: Three different distributions in the interval between 0 and 1

When ns is executed, it inserts a few initial events in the scheduler, then goes into a loop in which it searches the next event to be executed and executes its handler. The handler may do nothing related with the events, else it may insert some new events, or cancel any of the events which are currently in the scheduler. To reproduce this, we execute the ns simulator writing out all the operations related with the events performed in the execution of the handlers. We also wrote in the same file the initial events inserted in the scheduler. Our model is basically a

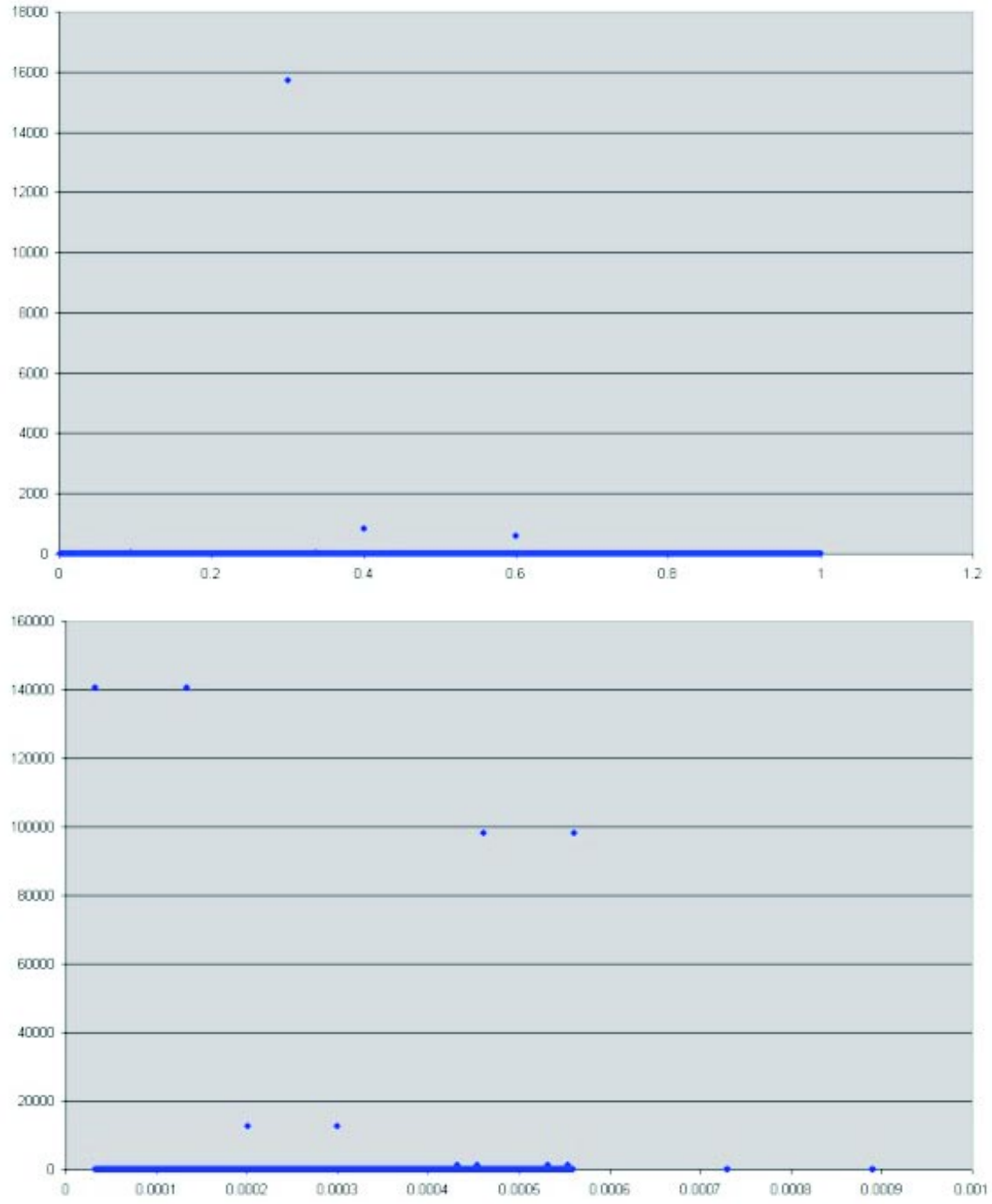


Figure 5: The peak points of the distribution

handler for the events, which reads what the actions associated with the execution of each event from a trace file. All the events in the execution of our model use our handler. In our model we introduced an initial event with 0.0 as the timestamp. In the execution of the handler the initial events are introduced, and after that the execution is exactly the same as the one in the real simulator, but now our handler is executed and the operations of the scheduler are read from a file. It is important to know that the trace file is load in a memory array at the begining of the

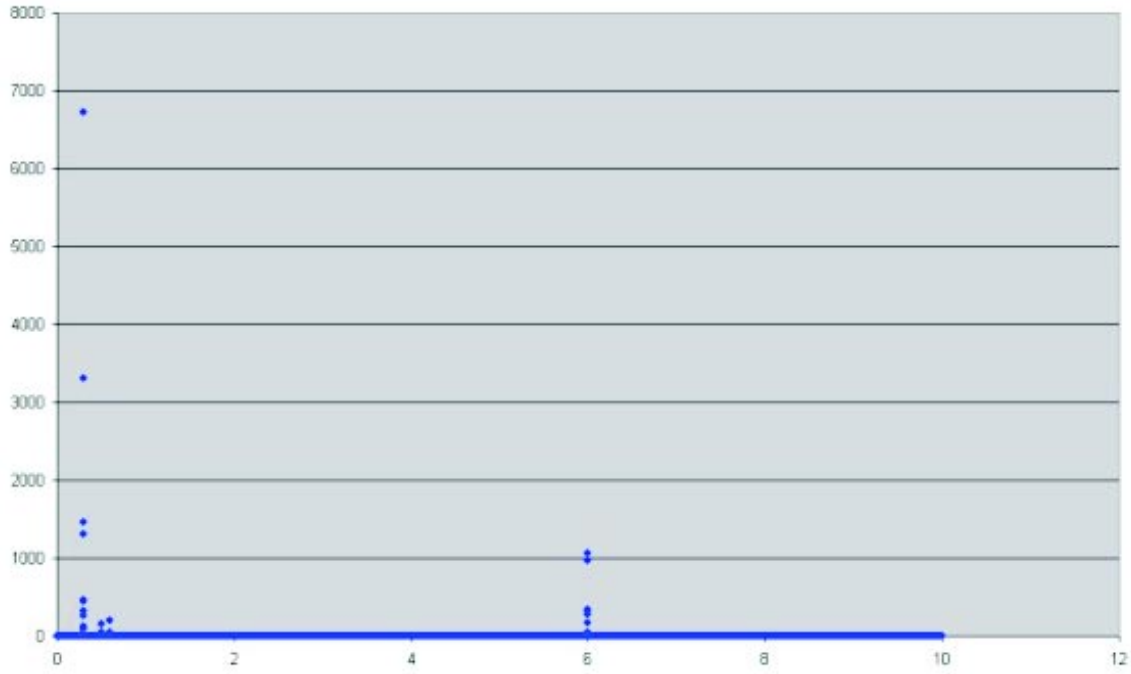


Figure 6: Delays cancellation distribution

model execution in order to avoid that the readings from the trace file may be measured as part of the application time modifying the results.

2.4 Our model in Java

When we had our model in C++ simulating the ns simulator, we translated it into Java. This translation was straight forward because they are both object oriented programming languages and the syntax between is very similar. We had no problems in this part of the project.

3 Results

At this moment we had the models implemented in Java and C++. The first thing that we had to decided was the size of the trace files, is that, the time we are going to run the simulations in order to get the trace files. We decided that it was interesting to run the simulations only 3 seconds because the trace files weren't so big and the simulation overcame the starting load. We decided this based in Figure **??**. At this point we also had to decided the interval in which we were going to take the measures. At the end we took the measures between 2 and 3 seconds.

All the executions were done in Sun HPC 6500 machine with 18 400MHz UltraSPARC-II processors, 18 Gbyte of shared memory and 90 Gbyte disc space. For C++ we used three different compilers: Sun Forte 6.1, Sun Forte 6.2 and gcc and for Java we exexuted our model in two different Java Virtual Machines: Sun JDK 1.2.1 and Sun JDK 1.3.1 with both client and server options. Figure **??** shows the execution time obtained for all of these compilers and Java Virtual Machines with trace files obtained from models with different number of nodes.

Figure 7: Time execution of our model with different sizes of the problem

4 Conclusion

The execution time is better in any of the JVMs than in C++. Figure 1 shows that the execution time of all the C++ compilers is very similar. The execution time in the different JVMs changes more, jdk1.3.1 with server option is the fastest one taking only the half execution time of the C++ model.

References

[1] Bull, J. M., Smith, L. A., Pottage, L. and Freeman, R., *Benchmarking Java against C and Fortran for Scientific Applications*, Proceedings of ACM 2001 Java Grande/ISCOPE Conference, Palo Alto, California, June 2001, pp. 97-105

[2] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty and R. A. Davey, *A Benchmark Suite for High Performance Java*, Concurrency, Practice and Experience, vol. 12, pp. 375-388, 2000

I'm Ana Bosque Arbiol. At this moment I'm student of Computer Engineering at Universidad de Zaragoza (Zaragoza's University).

My supervisors were Martin Westhead and Mark Bull

5 Appendix

5.1 More about ns

NS is an event driven network simulator developed at UC Berkeley that simulates variety of IP networks. It implements network protocols such as almost all variants of TCP and UPD, several forms of multicast, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra, and more.

The NS project is now a part of the VINT project. VINT is a DARPA-funded research project whose aim is to build a network simulator that will allow the study of scale and protocol inter-

action in the context of current and future network protocols. VINT is a collaborative project involving USC/ISI, Xerox PARC, LBNL, and UC Berkeley.

NS is Object-oriented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component object libraries, and network setup (plumbing) module libraries. To setup and run a simulation network, a user should write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting packets through the event scheduler. We use the term "plumbing" because setting up a network is plumbing possible data paths among network objects by setting the "neighbor" pointer of an object to the address of an appropriate object.

Another major component of NS beside network objects is the event scheduler. An event in NS is a packet ID that is unique in the system with scheduled time and the pointer to an object that handles it. In NS, an event scheduler keeps track of simulation time and fires all the events in the event queue scheduled for the current time by invoking appropriate network components. Network components communicate with one another passing packets, however this does not consume actual simulation time. All the network components that need to spend some simulation time handling a packet (i.e. need a delay) use the event scheduler by issuing an event for the packet and waiting for the event to be fired to itself before doing further action handling the packet. Another use of an event scheduler is timer.

When a simulation is finished, NS produces one or more text-based output files that contain detailed simulation data, if specified to do so in the input Tcl script. The data can be used for simulation analysis or as an input to a graphical simulation display tool called Network Animator (NAM) that is developed as a part of VINT project. NAM has a nice graphical user interface similar to that of a CD player and also has a display speed controller.

NS is available from http://www.isi.edu/nsnam/ns. From this site you can download it and also find further information about ns: how to install it, problems and bugs, tutorials at different levels and more.

Currently, NS (version 2) written in C++ and OTcl (Tcl script language with Object-oriented extensions developed at MIT) is available. In this project we began working with the 2.1b8a version, but for building the trace-files that we needed, we used the 2.1b5 version, because the Tcl scripts that we had didn't work properly in the other version.

5.2 Our first simulation

```
set ns [new Simulator]

set n1 [$ns node]
set n2 [$ns node]

$ns duplex-link $n1 $n2 10Mb 10ms DropTail
$ns queue-limit $n1 $n2 1000000
$ns queue-limit $n2 $n1 1000000

for {set i 0} {$i < 50} {incr i} {
set tcp($i) [new Agent/TCP/FullTcp]
$tcp($i) set fid_ $i
```

```
$tcp($i) set syn_ true
$ns attach-agent $n1 $tcp($i)
}

for {set i 0} {$i < 50} {incr i} {
set tcp($i_) [new Agent/TCP/FullTcp]
$tcp($i_) set fid_ [expr $i + 5]
$tcp($i_) set syn_ true
$tcp($i_) listen
$ns attach-agent $n2 $tcp($i_)
}

for {set i 0} {$i < 50} {incr i} {
$ns connect $tcp($i) $tcp($i_)
}

for {set i 0} {$i < 50} {incr i} {
$ns at 1.0 "$tcp($i) send 10485760"
}

$ns at 3600.0 "finish"

proc finish {} {
exit 0
}

$ns run
```

5.3 The next simulations

```
# number of nodes in the ring...  it changes from one simulation
to the other
#set num_nodes 8
# number of web clients per node (must be even)
set www_clients 2
# number ftp clients per node (must be even)
set ftp_clients 2
# number of nodes to servers (in each direction)
set server_dist 8
# bandwidth of connections
set bandwidth 10Mb
# delay of connections
set delay 0.1ms
# ftp file size (bytes) intended
# to require longer to transfer than
# simulation length
set filesize 100000000

# length of simulation
set stop_time 30
```

```
#TCP parameters
#Agent/TCP set window_ [expr $bdp*4]
set win_size 1
Agent/TCP set windowInit_ $win_size
set segperack 2
set delack 0.4
set lastsample 0
set client_addr 0
set no_of_inlines 0

########################################
# Initialisation

set ns [new Simulator]

set client_addr 0
ns-random 0
ns set-address-format expanded

########################################
# Topology

proc build_ring {ns} {
global delay bandwidth num_nodes node
set node(0) [$ns node]
for {set i 1} {$i < $num_nodes } {incr i} {
set node($i) [$ns node]
$ns duplex-link $node([expr $i -1]) $node($i) $bandwidth $delay Drop-
Tail
} $ns duplex-link $node([expr $i-1]) $node(0) $bandwidth $delay Drop-
Tail
}

proc clock_add { a b } {
global num_nodes
set a [expr $a set b [expr $b set c [expr $a + $b]
if { $c >= $num_nodes } {
return [expr $c - $num_nodes]
} else {
return [expr $a+$b]
}
}


proc clock_sub { a b } {
global num_nodes
set a [expr $a set b [expr $b set c [expr $a - $b]
if { $c < 0 } {
return [expr $c + $num_nodes]
} else {
return [expr $a - $b]
}
```

```
}

proc build_ftpclient {cnd snd sftp startTime timeToStop Flow_id}
{

global ns
global stopTime
set cli [get_ftpclient]
set ctcp [get_fulltcp]
$ctcp attach-application $cli
$ctcp set fid_ $Flow_id
$cli tcp $ctcp
$ns attach-agent $cnd $ctcp

set stcp [get_fulltcp]
$stcp attach-application $sftp
$stcp set fid_ $Flow_id
$ns attach-agent $snd $stcp

$ns connect $ctcp $stcp
$ctcp set dst_ [$stcp set addr_]
$stcp listen
$ns at $startTime "$cli start"
$ns at $timeToStop "$cli stop"
global ftplist
global ftplist
set ftplist($ctcp) $stcp
return $cli
}

proc build_wwwclient {cnd snd sp si startTime timeToStop Flow_id}
{

global ns stopTime no_of_inlines

set cli [get_wwwclient]
set ctcp [get_fulltcp]
$ctcp attach-application $cli
$ctcp set fid_ $Flow_id
$cli tcp-primary $ctcp
$ns attach-agent $cnd $ctcp

set stcp [get_fulltcp]
$stcp attach-application $sp
```

```
$stcp set fid_ $Flow_id
$ns attach-agent $snd $stcp


$ns connect $ctcp $stcp
$ctcp set dst_ [$stcp set addr_]
$stcp listen
global wwwlist
set wwwlist($ctcp) $stcp


for { set i 1 } {$i <= $no_of_inlines } { incr i 1 } {
set ctcp [get_fulltcp]
$ctcp attach-application $cli
$ctcp set fid_ $Flow_id
$cli tcp-in-line $ctcp
$ns attach-agent $cnd $ctcp


set stcp [get_fulltcp]
$stcp set fid_ $Flow_id
$stcp attach-application $si


$ns attach-agent $snd $stcp


$ns connect $ctcp $stcp
$ctcp set dst_ [$stcp set addr_]
$stcp listen
set wwwlist($ctcp) $stcp
}


$ns at $startTime "$cli start"
$ns at $timeToStop "$cli stop"


return $cli
}


proc get_ftpclient {} {


global client_addr
set cli [new Agent/TcpApp/FtpClient]
$cli set addr_ [incr client_addr]
return $cli
}


proc get_wwwclient {} {
```

```
global client_addr
set cli [new Agent/TcpApp/WWWClient]
$cli set addr_ [incr client_addr]
$cli max_think_time 1.0
return $cli
}


proc get_fulltcp {} {
global segperack delack
set atcp [new Agent/TCP/FullTcp]
$atcp set segsperack_ $segperack
$atcp set interval_ $delack
return $atcp
}


proc uniform {a b} {
expr $a + (($b- $a) * ([ns-random]*1.0/0x7fffffff))
}


build_ring $ns


for {set i 0} {$i < $num_nodes } {incr i} {
set j [clock_add $i $server_dist]
set g [clock_sub $i $server_dist]
if { [expr $ifor {set k 0} {$k < $ftp_clients} {incr k 2} {
set sftp1 [new Agent/TcpApp/FtpServer]
$sftp1 file_size $filesize
set sftp2 [new Agent/TcpApp/FtpServer]
$sftp2 file_size $filesize
build_ftpclient [set node($i)] [set node($j)] $sftp1  [uniform 0.0
1.0] $stop_time 2
build_ftpclient [set node($i)] [set node($g)] $sftp2  [uniform 0.0
1.0] $stop_time 2
}
} else {
for {set k 0} {$k < $www_clients} {incr k 2} {
set siwww1 [new Agent/TcpApp/WWWServer]
set spwww1 [new Agent/TcpApp/WWWServer]
$spwww1 primary
set siwww2 [new Agent/TcpApp/WWWServer]
set spwww2 [new Agent/TcpApp/WWWServer]
$spwww2 primary
build_wwwclient [set node($i)] [set node($j)] $spwww1 $siwww1  [uni-
form 0.0 1.0] $stop_time 3
build_wwwclient [set node($i)] [set node($g)] $spwww2 $siwww2  [uni-
form 0.0 1.0] $stop_time 3
```

```
}
}
}

$ns at $stop_time "finish"

proc finish {} {
exit 0
}

$ns run
```

5.4 A little more about our trace files

Here we explain a little more about the information recorded in the trace files and the format of the file. We discussed three different kinds of operations that can be executed over the scheduler; insertion, cancellation and search. The last of these is never executed in the models that we were simulating, but it is possible to execute it from a handler, so we designed our trace files in order that this operation could be put in them. We also needed a separator in order to know when the operations that an instance of our handler had to execute finished.

We put each instruction in a different line in our trace file. The first character in the line indicates the type of the operation. We use 'i' for insertion, 'c' for cancellation, 'b' for search and '*' as a separator. After that we write the information necessary for the performing of the operation: for the insertion the delay, for the cancellation the timestamp (it's the fastest way to locate the event) and the identification of the event, and for search the identification of the event. Below we show a piece of a trace file by way of example.

```
i 0.00013200000000002099
i 0.00003200000000003200
****
****
****
i 0.00013200000000002099
i 0.00003200000000003200
****
****
i 0.00056080000000002794
i 0.00046080000000003896
****
c 1.22343780681239722163 17023
i 0.00056080000000002794
i 0.00046080000000003896
i 0.30000000000000004441
****
****
i 0.00056080000000002794
```

```
i 0.00046080000000003896
****
```

In this section of a trace file there are the operations for 9 different handlers The first handler will execute two insertions, the two following handlers will execute nothing, the following two insertions and so on.